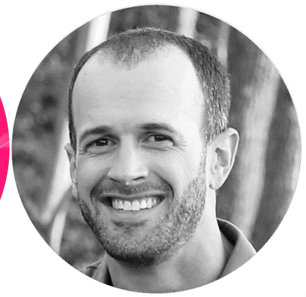
Managing Shared, Derived, and, Immutable State



Cory House

React Consultant and Trainer

@housecor | www.reactconsulting.com

Demo

Build shopping cart

- Derive state
- Explore where to declare state
- Lift state
- Immutability
  - Why bother?
  - Immutable friendly approaches
- Use function form of setState
- Declare event handlers on a list

Where to Declare State

Common mistake

Declaring state in the wrong spot

Sign

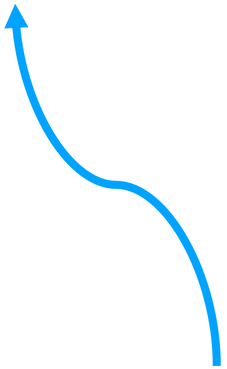
Hard to consume and share

Suggestion

Start local, and lift when needed

Principle of Least Privilege

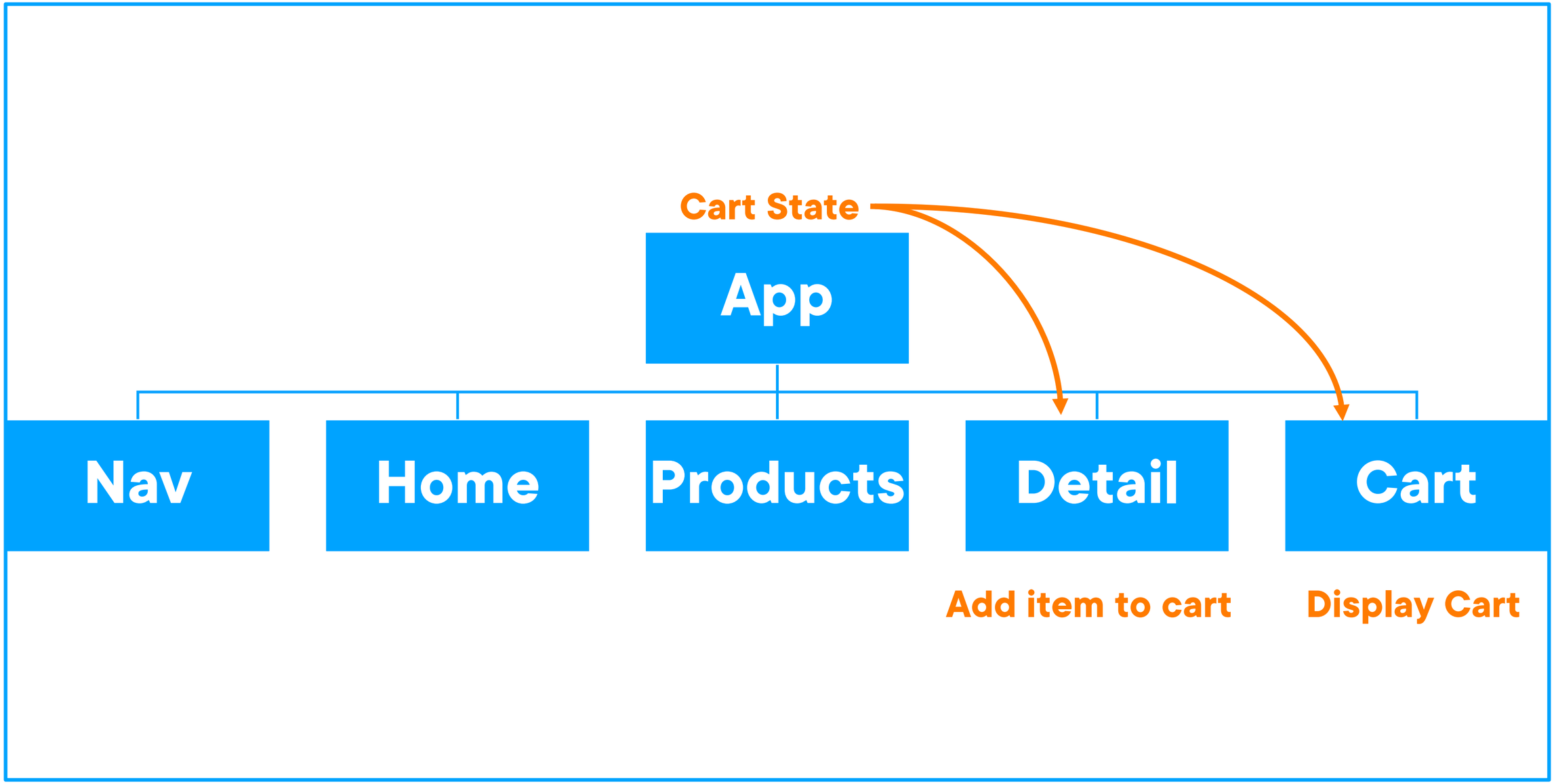
Every module must be able to access only the information and resources that are necessary for its legitimate purpose.



Ideally each React component only has access to the data/functions it needs

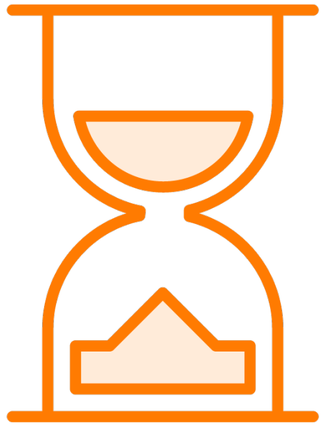
State: Start Local

1. Declare state in the component that needs it.
2. Child components need the state? Pass state down via props.
3. Non-child components need it? Lift state to common parent.
4. Passing props getting annoying? Consider context, Redux, etc.


Cart State
App
Nav
Home
Products
Detail
Cart
Add item to cart
Display Cart

Why Is Setting State Async?

Batching



Supports async rendering

Use Function Form to Reference Existing State

```
const [count, setCount] = useState(0);

// Avoid since unreliable
setCount(count + 1);


// Prefer this — Use a function to
reference existing state
setCount((count) => count + 1);
```

Why Immutability?

Fast comparisons

Pure funcs are easy to understand and test

Simpler undo/redo

Avoids bugs

Immutability = Performance

```
user = {
  name: 'Cory House'
  role: 'author'
  city: 'Kansas City'
  state: 'Kansas'
  country: 'USA'
  isFunny: 'Rarely'
  smellsFunny: 'Often'
  ...
}
```

Value vs. Reference Equality

```
user = {
  name: 'Cory House'
  role: 'author'
  city: 'Kansas City'
  state: 'Kansas'
  country: 'USA'
  isFunny: 'Rarely'
  smellsFunny: 'Often'
  ...
}
```

```
const user1 = user;
const user2 = user;
```

Value equality

Does each property have the same value?

```
user1.name === user2.name &&
user1.role === user2.role &&
...
```

Reference equality

Do both vars reference the same spot in memory?

```
user1 === user2
```

Fast. Scalable. Simple. 👍

```
if (prevState !== state) ...
```

Avoid unnecessary re-renders

Function	**React.memo,**
Class	**shouldComponentUpdate, PureComponent**

Immutability:

To change state, return a new value.

What's Mutable in JavaScript?

Immutable already	VS	Mutable
Number		Objects
String		Arrays
Boolean		Functions
Undefined		
Null		

```
state = {
  name: 'Cory House'
  role: 'author'
}
```

◀ Current State

```
state.role = 'admin';
return state;
```

◀ Mutating State

```
state = {
  name: 'Cory House'
  role: 'author'
}
```

◀ Current State

```
return state = {
  name: 'Cory House'
  role: 'admin'
}
```

◀ Not Mutating State

Handling Immutable Data in JavaScript

Object.assign

Spread syntax

Some array methods

Copy via Object.assign

```
Object.assign({}, state, { role: "admin" });
```

Create an empty object…

Then add these properties

Copy via Spread

```
const newState = { ...state, role: "admin" };
```

```
const newUsers = [...state.users];
```

We'll use this since it requires less code than Object.assign

Warning: Shallow Copies

```
const user = {
  name: 'Cory',
  address: {
    state: 'California'
  }
}
```

```
// Watch out, it didn't clone the nested address object!
const userCopy = { ...user };
```

```
// This clones the nested address object too
const userCopy = { ...user, address: {...user.address}};
```

Avoid Nested Objects

```
const user = {
  name: "Cory",
  email: "cory@reactjsconsulting.com",
  address: {
    city: "Chicago"
  }
}

// Avoid
const [user, setUser] = useState(user);


// Prefer
const [user, setUser] = useState(user);
const [address, setAddress] = useState(user.address);
```

Warning: Only Clone What Changes

You might be tempted to use deep merging tools like clone-deep, or lodash.merge, but avoid blindly deep cloning.

Here's why:

1. Deep cloning is expensive
2. Deep cloning is typically wasteful
3. Deep cloning causes unnecessary renders

Instead, clone only the sub-object(s) that have changed.

Handling Arrays

Avoid		VS	Prefer
push	Must clone array first		map
pop			filter
reverse			reduce
			concat
			spread

Handling Immutable State

Native JavaScript

Object.assign

Spread operator

Map, filter, reduce

Libraries

Immer

seamless-immutable

react-addons-update

Immutable.js

Many more

```
const numItemsInCart = cart.reduce((total, item) => total +
item.quantity, 0);
```

```
const numItemsInCart = useMemo(
  () => cart.reduce((total, item) => total + item.quantity, 0),
  [cart]
);
```

Web Storage

Cookie

localStorage

sessionStorage

IndexedDb

Cache storage

Web Storage Tradeoffs

Benefits	VS	Downsides
Local		Limited storage
Simple		Security risk
Fast		localStorage / sessionStorage block I/O
Works offline		Tied to single browser

Example

The following snippet accesses the current domain's local `Storage` object and adds a data item to it using `Storage.setItem()`.

```
1 | localStorage.setItem('myCat', 'Tom');
```

The syntax for reading the `localStorage` item is as follows:

```
1 | var cat = localStorage.getItem('myCat');
```

The syntax for removing the `localStorage` item is as follows:

```
1 | localStorage.removeItem('myCat');
```

The syntax for removing all the `localStorage` items is as follows:

```
1 | // Clear all items
2 | localStorage.clear();
```

Note: Please refer to the Using the Web Storage API article for a full example.

Summary

Derive state when possible

Lift state to share it

Treat React's state as immutable

- Use spread, map, filter, etc.

Setting state is async and batched

- Use a func to reference current state

Persist state to localStorage via useEffect

- Many web storage options

Up Next:

Form Validation